ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Translating Pedagogical Verification Exercises to Viper

Bachelor's Thesis

Benjamin Frei

Sunday 17th September, 2023

Advisors: Nicolas Klose, João Pereira
Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Contents

Abstract

Viper is a verification language designed by the Programming Methodology Group of the Department of Computer Science at ETH Zürich. Program verification languages are designed to verify the specifications of programs. With the help of verifiers, we can ensure the correct behaviour of critical applications.

Program verification is hard. Currently, there is a lack of good pedagogical material for beginners, entering the field of program verification. This thesis creates new pedagogical material. This material is aimed at beginners in program verification. We will start by translating examples and exercises from the Book Program Proofs [1] written in the verification language Dafny, to Viper.

Chapter 1

Introduction

Today's world wouldn't be imaginable without software. There is an enormous amount of software being designed and implemented for very different systems. For many applications, it would be valuable to guarantee that programs will behave as they are supposed to. Rather than relying solely on the testing of a program, with the help of verification languages such as Viper [2] and Dafny [3], we are able to prove the correctness of such programs with respect to their formal specification.

Verification languages are programming languages that focus on ensuring the correctness, reliability, and security of software through formal verification. These languages often come with tooling, and with the help of those tools we are able to prove that code satisfies certain specifications. Verification languages focus on the verification of program properties, such as safety, liveness, termination, and absence of vulnerabilities. They often employ formal logics, such as first-order logic, temporal logic, or separation logic, to express program specifications, assertions, preconditions, postconditions, and loop invariants. These specifications serve as a basis for formal reasoning and verification.

Verification languages typically provide tools that perform automated analysis on programs written in these languages. These tools use techniques like model checking, abstract interpretation, symbolic execution, and theorem proving to verify the desired properties of the code. Viper [2] and Dafny [3] use multiple such concepts to enable a user to write verified code.

As we see in Figure 1.1, Viper consists of the Viper intermediate language, the symbolic execution backend, as well as of the Verification Condition generation backend. There are frontends that transform code written in other languages into Viper (such as Gobra [4] for Go or Nagini [5] for Python). After this transformation, they can use the Viper infrastructure to prove the correctness of the translated programs, giving them the certainty that their

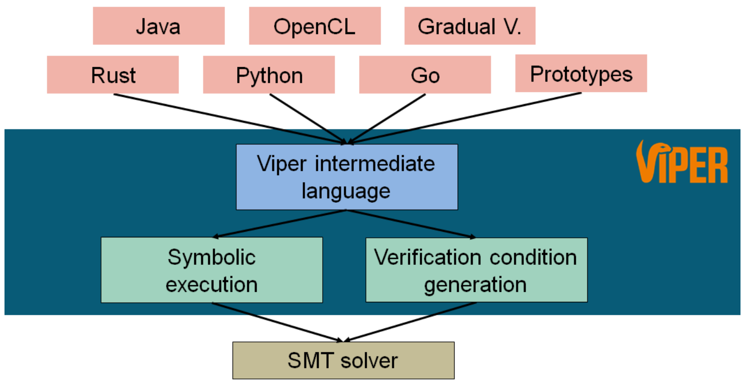

Figure 1.1: This figure shows how Viper is used to verify code written in other languages, by translating programs in these languages to a Viper program. [2]

code is correct.

Program verification is hard, and it is very helpful to be able to look at and analyse code. The motivation for this project is to create pedagogical material that introduces Viper to programmers without previous experience in program verification. The book Program Proofs [1] will be used as a source of examples and exercises for pedagogical material in Viper. The book Program Proofs is aimed at programmers without any experience in program verification, it serves as an introductory textbook on program verification. The book uses the programming language Dafny in all of its examples and exercises. The book focuses on the basic features of program verifiers, such as how to specify behaviors of functions and methods using preconditions and postconditions and how to deal with loops and recursive functions in program verification. All the basic features of Dafny are further covered in the Backround section of this thesis. To create new pedagogical material for Viper, this thesis translates examples and exercises from this book and documents the most important differences between the Dafny and Viper versions in a way that is easy to understand for beginners of both languages. The thesis covers the chapters 6 to 8 and the chapters 10 to 12 of the book. This material should give an intuition for most concepts provided by a verification language.

Chapter 2

Background

This chapter contains information about the different materials and program verification languages used in this thesis. First, we will give a quick overview of the book Program Proofs and we will discuss the topics contained in the book. Secondly, we will take a look at the program verification language Dafny and Viper. Dafny which is used in the book and Viper the language for wich we will be creating pedagogical material. We will be looking at features used and supported by Dafny and Viper.

2.1 The Book: Program Proofs

The book was written by K. Rustan M. Leino and published in March 2023. The author is a Senior Principal Applied Scientist in the Automated Reasoning Group at Amazon Web Services. It contains 18 Chapters and is divided into three parts. Each of the 18 chapters introduces us to features of the verification language Dafny.

The three parts of the book are Part 0: Learning the Ropes, Part 1: Functional Programs, and Part 2: Imperative Programs. Chapter 0 provides us with a quick introduction. It introduces program verification in general and establishes the prerequisites needed in order to understand the examples and exercises of the book, it gives a quick overview of the topics covered in the book and introduces the verification language Dafny.

Part 0

Part 0 of the book was skipped, this is because there already exists a translation for this part. We do not translate any code covered in this part to Viper. Nevertheless, it is still important to quickly mention its contents, since it introduces features used in chapters 6 to chapter 12 which we are translating.

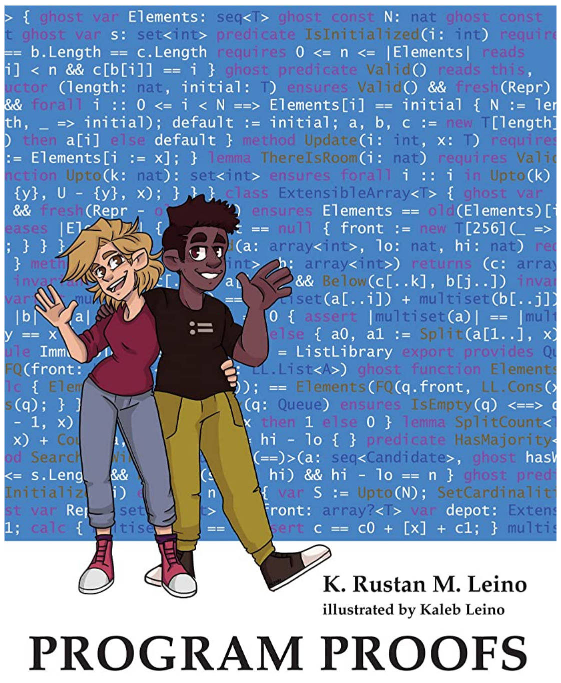

Figure 2.1: This figure shows the cover of the Book Program Proofs [1]

Part 0 includes chapters 1 to 5. Part 0 deals with the basic features of Dafny such as Methods, Assert Statements or Compiled vs Ghost code, introducing Hoare logic, recursion and termination, inductive data types to lemmas and proofs in Dafny.

Part 1

The second part of the book deals with more specific topics. Titled Functional Programs, the features introduced and used are similar to the ones used in functional programming, defined by the use of recursion and match statements. In this thesis, we will not cover Chapter 9. This is because Chapter 9 is dealing with modules. In Dafny this is done by declaring a module. Functions in modules can be accessed and seen only by other functions inside the module unless they are exported. Viper does not support anything similar to modules, and it does not make sense to translate the examples and exercises that explore these features.

In this part, all the chapters except chapter 9 were translated from Dafny to Viper, therefore we will later give a bit more details about them.

Part 2

The third and final part of the book deals with imperative programming. It contains chapters 11 up to chapter 17. For this thesis, we only translated chapters 11 and 12.

Chapter 11 introduces loops and loop invariants. It teaches us how to find loop invariants. It also concerns itself with the termination proof of loops using decreasing values in each loop iteration. Chapter 12 deals with recursive specifications and iterative programs, like iterative Fibonacci. It elaborates on differently optimized implementations of the Fibonacci sequence as well as of the power function.

2.2 Features

This section introduces the main features of Dafny and Viper. In general, Dafny has more high-level constructs as Dafny was designed for programmers to write their applications in Dafny, while also verifying them in Dafny. On the other hand, Viper was designed as an intermediate verification language, meaning it is more often used as an interface to verify code written in other languages.

Dafny

Dafny is a verification-aware programming language and verification tool designed to help write correct and reliable programs. It was developed at Microsoft Research and is primarily used for program verification. Dafny is based on the feature of programming by contract, where programs are annotated with preconditions, postconditions, and invariants that define their behavior. These specifications can be used to specify the intended properties of the program, and the Dafny tool can then automatically verify if the program meets those specifications.

Section 2.2 introduces the features of Dafny. We will show the different features provided by Dafny. Throughout this section, we will mention, whether there is an equivalent feature supported by Viper. We try to mention the features in the order they appear in the book.

Viper

Viper is a verification language and verification tool that is designed for program verification. Viper is not a compiled language and therefore, cannot be compiled and executed contrary to Dafny. The intermediate verification language Viper was developed by ETH Zürich. It is currently used for educational purposes by many other institutions, such as Brown University and Columbia University.

Many verifiers are built on top of Viper, such as Gobra for Go, Nagini for Python, or Prusti for Rust. Those frontends are tools that allow developers to automatically create Viper encodings of their code. Those encodings will then be analysed by Viper and its formal verification tools. This helps to ensure that the code behaves correctly according to its specifications.

Viper backends

At this point, it makes sense to mention that there are two different backends for Viper. One is called Viper Silicon, and it is set as the default verifier in Visual Studio Code, the other verifier is called Carbon [6]. Silicon is a symbolic-execution-based verifier for the Viper intermediate verification language, while Carbon is a verification-condition-generation-based verifier. The performance of such program verifiers based on SMT solvers is often unpredictable, and in some cases, one solver might perform better than the other depending on the program.

Symbolic-execution-based verifier is a formal verification tool or approach that uses symbolic execution to analyze and reason about programs for correctness. Symbolic execution is a technique used in software analysis and verification where program execution is performed symbolically, representing program inputs and variables as symbolic expressions rather than concrete values.

In symbolic execution program paths are explored symbolically considering different possible values and branches that the variables can take during execution. Instead of executing the program with actual input values, the verifier works with abstract interpretations of inputs and tracks how the program's variables and expressions change based on these symbols.

This type of verifier checks whether each branch of the program is safe and generates multiple smaller queries per method to the underlying solver. In the case of Viper, this solver is Z3 [7]. [8]

Verification-condition-generation-based verifier generates verification conditions to check the correctness of a program. Verification condition generation is a common technique used in formal methods and program analysis to automatically generate logical conditions that capture the correctness properties that need to be verified.

Contrary to the Symbolic-execution-based verifier behind Silicon, a verification condition generation based verifier generates a single larger query per method and sends it to the underlying solver. Both Viper backends Silicon and Carbon use the Z3 solver [7]. [9]

Methods

In Dafny and Viper, a method refers to a sequence of code that performs a specific task or computation. It is a fundamental building block of programs written in the Dafny and Viper programming language. Methods consist of a series of instructions and can take input parameters, perform operations, and return results.

Methods can have various combinations of input parameters, local variables, loops and conditionals, and other programming constructs. They can perform computations, call other methods, and interact with objects and data structures.

```
method Triple(x: int) returns (r: int)
    requires x % 2 == 0
    ensures r == 3 * x
{
  if x == 0 {
    r := 0;
  } else {
    var y := 2 * x;
    r := x + y;
  }
  assert r == 3 * x;
}
```

Figure 2.2: This is an example of a method written in Dafny. It uses precondition, postcondition, if and else branching as well as assignments and assert statements. [1]

```
method Triple2(x: Int) returns (r: Int)
    requires x % 2 == 0
    ensures r == 3 * x
{
  if (x == 0) {
    r := 0
  } else {
    var y : Int := 2 * x;
    r := x + y;
  }
  assert r == 3 * x;
}
```

Figure 2.3: This code example shows the same method as in Figure 2.2 written in Viper. The syntax is very similar. The biggest difference in this example is the mandatory brackets around the if condition, as well as the forced declaration of the variable type of y.

Methods can also have preconditions and postconditions specified using Dafny's or Vipers contract annotations seen in section 2.2. Preconditions describe the assumptions that must hold before the method is invoked, while postconditions specify the expected properties or guarantees that should hold after the completion of the method.

Functions

Functions are blocks of code that take input parameters, perform computations, and return a single value. A function in Viper and Dafny corresponds to a mathematical function. Function bodies consist of a single expression.

Functions and methods look similar. They are however not the same. Functions do not have side effects and their return value is solely determined by their inputs. This enables us to use functions in preconditions and postconditions to verify properties. Methods on the other hand are not side effect-free and might return different values given the same input.

```
function Length'<T>(xs: List<T>): nat {
  if xs == Nil then 0 else 1 + Length'(xs.tail)
}
```

Figure 2.4: This code 2.4 shows the declaration of a function in Dafny.

```
function length(xs : List[Int]) : Int
  ensures result >= 0
  decreases xs
{
  xs.isNil ? 0 : (1 + length(xs.tail))
}
```

Figure 2.5: In this code example 2.5 we see the equivalent declaration of the function Length' written in Dafny in Figure 2.4. In Viper, a function can only consist of expressions. If and else conditions are not categorized as expressions in Viper and can therefore not be used in function bodies. Due to this, the ternary operator ? was used.

Assert Statements

Assert Statements allow you to specify and enforce certain conditions or properties within a program. Assert statements are used for runtime checks and can help identify potential errors or inconsistencies.

Assert statements are helpful for expressing expected properties within the program. They act as sanity checks during program execution, helping to catch potential bugs or incorrect program states, by asserting statements that are expected to be true. In some cases, assert statements can help prove certain statements, by making Dafny or Viper aware of additional facts that help the verifier in proving a postcondition.

An example of an assert statement in Dafny can be seen in this code example fig. 2.2. The same syntax is also used in Viper, as seen in code this code example fig. 2.3.

If, Else and Match statements

Dafny supports if and else statements, and pattern matching. Viper only supports if and else statements, but it does not support pattern matching. If and else statements in Dafny are seen as expressions as they can be used in functions. In Viper, they are not treated as expressions but are statements and can not be used in functions. If a method contains such control path statements, Dafny or Viper will prove a postcondition by proving it for all paths.

Match statements are equivalent to the ones used in functional programming languages and can be compared to if, else, and else if branches with the conditions regarding the matched structure.

An example of if and else branching can be seen for Dafny in the code example 2.2 and for Viper in 2.3. The syntax for a match statement in Dafny can be seen in the code example 2.6.

Preconditions, Postconditions and Loop Invariants

In Dafny and Viper, preconditions, postconditions, and loop invariants are contract annotations used to specify properties and conditions of the program's behavior. These annotations help to formally verify programs, enabling Dafny or Viper to statically analyze and prove the correctness properties of a program.

Preconditions are used to specify the assumptions or requirements that must hold before a method is invoked or a loop iteration begins. They define the conditions under which the method or loop is expected to behave as intended. Preconditions are specified using the ”requires” keyword in Dafny and Viper.

Postconditions describe the expected properties or guarantees that a method should satisfy upon completion. They specify the conditions that must hold after the execution of a method or a loop. Postconditions are specified using the ”ensures” keyword in Dafny. An example of a precondtion and postcondition can be seen in Dafny and Viper in example 2.2 and example 2.3, respectively.

Loop invariants are used to specify properties that hold before and after each iteration of a loop. They are crucial for loop termination and ensuring loop correctness. Loop invariants are specified using the "invariant" keyword in Dafny. Example 2.10 and example 2.9 show the application of an invariant in Dafny and Viper, respectively.

By specifying preconditions, postconditions, and loop invariants, developers can express the expected behavior of their code in a formal and verifiable way, enabling the verifier to prove the correct execution of the program.

Ghost Code

Ghost code is denoted by the "ghost" keyword in Dafny, which indicates that the code is intended for verification purposes only and should be ignored during compilation. The main purpose of ghost code is to provide additional assertions, or auxiliary functions, that help in the verification of a program. It allows programmers to express properties and invariants that cannot be expressed in executable code, or properties that do not need to be in the executable code.

```
ghost function Elements(pq: PQueue): multiset<int> {
    match pq
    case Leaf => multiset{}
    case Node(x, left, right) =>
      multiset{x} + Elements(left) + Elements(right)
}
```

Figure 2.6: This is a program that shows us an example of Dafny ghost code. This function calculates the number of elements stored in a PQueue recursively. This program is only used for specification and verification purposes and is therefore flagged as ghost code. [10]

Ghost code is a powerful feature in Dafny that simplifies formal verification and supports precise program specifications and properties beyond what is captured in the executable code. Ghost code enables a programmer to introduce additional code that aids verification, while not compromising the speed and complexity of the compiled code.

Viper does not support ghost code. Viper was designed as an intermediate verification language. As such, Viper provides only a very limited set of features. Nonetheless, ghost code is useful and most of the frontends of Viper do support ghost code. Implementing ghost code is thus the responsibility of the developers of the front-ends.

Simultaneous assignments

Dafny supports simultaneous assignments. This allows to assign multiple variables to values simultaneously within a single statement. This feature enables efficient assignment of values to multiple variables in a single operation. Simultaneous assignments are particularly useful when performing operations involving multiple variables that need to be updated together, such as swapping values or parallel assignments.

```
x, y := y, x;
```

Figure 2.7: This shows the simultaneous assignment of two variables in Dafny. In this code example the values of x and y are swapped.

Viper does not support simultaneous assignments. Unfortunately, we cannot simply translate statements like the one in Figure 2.7 from Dafny to Viper. We need an intermediate step in order to save the first variable assignment. The respective code in Viper would look like the code in Figure 2.8.

```
x = temp

x = y

y = temp
```

Figure 2.8: This shows how the equivalent code to the simultaneous assignment in Dafny seen in Figure 2.7 looks like in Viper.

Termination

Termination is a crucial aspect of program correctness and is necessary for Dafny's verification process. In Viper, termination proofs are not enforced. Nonetheless, the user is still able to prove termination by providing a termination measure in a "decreases" clause. The user can still prove termination by providing a termination measure. Dafny and Viper statically analyses the code to ensure that every loop and recursive function terminates. It checks whether the termination condition is bounded and decreasing across recursive calls. If no termination measure is provided Viper defaults to a constant. This usually suffices for non-recursive functions without loops.

Termination is crucial because it guarantees that the program will eventually complete its execution and not get stuck in an infinite loop. To help ensure termination Dafny and Viper require loop invariants and recursive functions to have a clear termination argument or measure. Dafny generates a termination condition automatically. If it is not able to prove termination with the automatically generated condition, one can be manually specified using the keyword "decreases".

```
method Loops11200()
    decreases
{
    var x : Int
    var y : Int

    x := 0
    y := 191
    while(7 <= y)
        invariant 0 <= 7 && 7 * x + y == 191
        decreases y
    {
        y := y -7
        x := x + 1
    }
}
```

Figure 2.9: This code shows a Viper method, where the decreases clause is used in order to force Viper to check for termination of the loop and the function.

In contrast to Dafny, Viper does not automatically generate a termination criterion. We have to provide a termination criterion to the function in order to check the termination of methods or functions.

```
method DivMod7() {
  var x, y;

  x, y := 0, 191;
  while 7 <= y
    invariant 0 <= y && 7 * x + y == 191
    decreases y
  {
    y := y - 7;
    x := x + 1;
  }
}
```

Figure 2.10: This next example 2.10 shows the equivalent method to method Loops11200 in Dafny. Here, we do not have to provide a "decreases" clause at the start of the method, since Dafny checks for termination automatically. However, in order to check for loop termination in Dafny we have to append a "decreases" clause, as seen in example 2.10.

In example 2.9, we see a method that is proven to terminate. In some easy cases Viper is able to infer the termination condition itself. We do not need to specify any condition for the method Loops1120, since the method simply checks whether its body terminates. The while loop inside the method has a specified termination criterion, namely y decreases with each iteration. With this information Viper proves the termination of the method.

Algebraic Data Type

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

Figure 2.11: This code shows how an ADT can be declared in Dafny. [1]

Algebraic Data Types (ADTs) in Dafny can be created using the syntax shown in Figure 2.11. ADTs are a feature in computer programming that originated from functional programming languages like Haskell. ADTs allow you to define composite data types by combining existing types using sum types and product types. Dafny supports generic ADTs, meaning we do not have to specify the type of the list element as seen in Figure 2.11 by introducing T as a generic type.

The example given in Figure 2.11 shows a sum type, where the Nil is the constructor for the base case and Cons is the constructor for the elements of this data type.

```
adt List[T] {
  Nil()
  Cons(value : T, tail : List[T])
}
```

Figure 2.12: This code shows how an ADT is defined in Viper.

Viper also supports Algebraic Data Types. In Figure 2.12 we see how such a type would be defined in Viper. However, Viper does not support generic functions using such Algebraic Data Types. Viper also does not automatically generate a well-founded order over the Algebraic Data Type.

Lemmas

Lemmas are auxiliary assertions or properties that can be used to state and prove additional facts about a program. Lemmas serve as intermediate steps in the formal verification process to establish certain properties or invariants.

Lemmas are used to express and prove intermediate results that are useful for reasoning about the correctness of a program or method but are not necessary for the program's execution. They can be used to establish invariants, derive properties, or simplify complex proofs. They enable modular reasoning and allow developers to establish and reuse intermediate facts.

```
lemma ReverseAuxAppend<T>(xs: List<T>, ys: List<T>, acc: List<T>)
  ensures ReverseAux(Append(xs, ys), acc)
       == Append(Reverse(ys), ReverseAux(xs, acc))
{
  match xs
  case Nil =>
    ReverseAuxCorrect(ys, acc);
  case Cons(x, tail) =>
}
```

Figure 2.13: This figure shows us an example of a lemma in Dafny. [10]

Lemmas in Dafny are theorems that are used to prove a result. Lemmas do not have a goal in and of themselves and are only useful for verification. They allow Dafny to break the proof of a more complex statement into multiple parts proving each of those parts separately. In the end, those proofs are combined to finally prove the correctness of the program or function. Viper does not have a construct called lemma. We model lemmas in Viper as described in section 3.1.

Automatic Induction

Automatic induction refers to a feature supported by Dafny that automates the process of applying inductive reasoning to prove properties of recursive functions and data structures, such as the List ADT created in Figure 2.11.

Dafny's automatic induction feature uses the structure of recursive functions and data types to automatically generate induction proofs. By automating induction, Dafny reduces the complexity and effort for the programmer required to prove the properties of recursive structures. There is also a limit to automatic induction in Dafny and there do exist cases, where the proof has to be done manually by the programmer.

```
lemma {:induction false} AddZero(x: Unary)
  ensures Add(Zero, x) == x
{
  match x
  case Zero =>
  case Suc(x') =>
    calc {
      Add(Zero, Suc(x'));
    ==  // def. Add
      Suc(Add(Zero, x'));
    ==  { AddZero(x'); }
      Suc(x');
    }
}
```

Figure 2.14: This code shows a lemma with induction false and a calc block to calculate the proof. Here the induction proof has to be done manually. [10]

Unlike Dafny, Viper does not support automatic induction. This can be seen in the translated examples as we have to specify the induction steps Viper needs to verify a statement each time. If turned off automatic induction in Dafny using the {*: induction false*} specification, we would have to state the induction step and create similar proofs to the ones in Viper.

```
function lemmaSnocAppend(xs : List[Int], y: Int) : Unit
  ensures Snoc(xs, y) == append(xs, Cons(y, Nil()))
  decreases xs
{
  (xs.isNil) ? unit() : lemmaSnocAppend(xs.tail, y)
}
```

Figure 2.15: This program shows a Viper method where the decreases clause is used in order to force Viper to check for the termination of the loop and the function.

Calc Blocks

Calc blocks in Dafny are sequences of logical or mathematical steps that help the verifier prove certain conditions. A calc block enables the programmer to write down the derivation steps to help the verifier prove the correctness of the code. An example of a calc block in Dafny can be seen in Figure 2.14. Viper does not support such a calc block.

Let expressions

In Dafny and Viper, let expressions are used to introduce local variables into the code body. It allows you to define and bind values to variables that are only valid within the scope of the let expression. This is mainly done to simplify function bodies and to be able to reuse local variables for expressions appearing multiple times inside the function.

"Let" expressions in Dafny and Viper allow developers to define temporary variables for intermediate calculations or to bind values to names for clarity and reuse within a limited scope. They can improve the readability of code by providing a way to structure complex expressions or calculations.

Domain

Domains in Viper allow for the introduction of additional types, mathematical functions, and axioms that define their properties. In terms of syntax, a domain consists of a name and a block where several function declarations and axioms can be defined. The Figure 1 contained in the appendix illustrates a basic domain declaration.

The functions declared within a domain have a global scope, meaning they can be applied anywhere else within the Viper program. These functions, known as domain functions, have certain limitations compared to the standard Viper functions discussed earlier. Specifically, domain functions cannot have preconditions. Additionally, they are always abstract, meaning they lack a defined body. The behavior of those functions is usually described by axioms.

Domain axioms also possess a global scope as they define properties that hold in all program states. As axioms must be well-defined across all states they cannot reference the values of heap locations or permission amounts. Domain axioms are expressed as standard first-order logic assertions they often use quantifiers to state their properties. [11]

Dafny does not support a domain construct. The closest Dafny construct to domains is Objects. Objects are however discussed in chapter 17 and are not relevant for this thesis.

Chapter 3

Translation

In this chapter, we dive into the translation of each of the chapters. We discuss the key points of those chapters and the problems faced during their translation. We also look at how the features in Dafny were translated to Viper, especially those that do not exist in Viper.

3.1 Translated Features

In this section, we will go in-depth on how we model features or types in Viper that do not have an equivalent to the ones in Dafny.

Lemmas

After starting the translation of the exercises, it is clear that an equivalent construct to the lemmas in Dafny had to be found. In order to model the lemmas as closely as possible, two main questions had to be answered.

- Do we implement lemmas as a function or as a method?
- If we implement lemmas as functions, what return type will they use?

We choose to translate lemmas as functions that return a type Unit, which was defined by an ADT that consists of a single possible value. The return type Unit was created in order to differentiate between normal functions and lemmas, as lemmas are not supposed to have a return type. An alternative we considered was to create lemmas as methods where we would not need to specify a return type. We would also be able to use if and else conditions in methods, making them more readable than with ternary operators. We stepped away from this approach since we could not use methods in function preconditions or postconditions, making it impossible to prove certain conditions. An example of such a lemma created in Viper is seen in Figure 2.14. Another important reason for implementing lemmas as functions is because otherwise, we would not be able to call lemmas from function bodies. This would make it impossible to prove a number of postconditions.

```
adt Unit {
  unit()
}
```

Figure 3.1: This is the type Unit defined in Viper.

Well-founded order on user-defined Data Types

While translating the examples, the problem of how to prove the termination of our newly created ADT arose. Contrary to Dafny, the ADT functionality in Viper does not automatically generate a well-founded order for its data type. In order to prove the termination of our functions we need to define a well-founded order for our data structure. With this order Viper can define which structures of the data type are structurally smaller and can prove the decreasing size of function arguments on recursive calls.

This code in Figure 3.2 shows how the well-founded order on the List[T] data type has been defined. *decreasing()* and *bounded()* are both functions defined in the file decreases/all.vpr. This file is part of the standard library of Viper and includes the well-founded orders of all data types initially defined by Viper, such as Int or Rational. *decreasing()* defines the structurally smaller element of a user-defined data type created by a domain, such as List[T] in our example. The *bounded()* function defines the smallest possible element of a user-defined data type. In our case this is the *Nil()* construct of List[T]. With those two functions we are able to create a well-founded order on any user-defined data type.

```
import <decreases/all.vpr>
/**
 * The ADT plugin of Viper generates an
 * Abstract Data Type, but it does
 * not automatically construct a well-founded
 * order regarding this Data Type.
 * Therefore we have to define a well-founded
 * order ourselves. This order is needed
 * to prove the termination of functions and methods.
 *
 * To define such a well-founded order, we use
 * a construct supported by Viper called
 * domain. A domain is used to define a new
 * type, by introducing functions and
 * using axioms to describe those functions
 * (We will see another example of a domain
 * later). In this example, we only use the
 * axioms to describe the behavior of well
 * founded order over the adt List.
 *
 * We chose to define our well-founded order
 * over the structural inclusion of a list.
 * This means that, for all lists x and y, x
 * is smaller than y if and only if x is
 * a suffix of y. This is not the only way of
 * defining a well-founded order. We
 * could have also defined it over the length of a list.
 */
```

Figure 3.2: This code shows the definition of the well-founded order on the List[T] data type. This is the first part, the second part can be seen in Figure 3.3.

```
domain ListWellFoundedOrder[T] {

  /**
   * This axiom establishes that, for each
   * variable of type List[T], there
   * exists a minimal element. In our case,
   * this minimal element corresponds to
   * Nil(). The function bounded is defined in
   * the imported file
   * (import <decreases/all.vpr>). The
   * bounded(y) in curly brackets is the trigger, this
   * axiom will be triggered whenever the
   * function bounded(y) for some y of type List[T]
   * is called.
   */
  axiom {
    forall y : List[T] :: {bounded(y)} bounded(y)
  }

  /**
   * With this axiom and the axiom above, we
   * define an order. The decreasing function
   * is defined in the file
   * <decreases/all.vpr> just like the bounded function. For
   * decreasing(xs, Cons(y, xs)) it defines xs
   * to be smaller than Cons(y, xs) regarding
   * the order of a List[T]. Unlike above, we
   * have not defined a trigger for this
   * axiom, Viper will automatically define such a trigger.
   */
  axiom {
    forall xs : List[T] , y : T ::
      decreasing(xs, Cons(y, xs))
  }

  /**
   * This axiom defines the transitivity of
   * the well-founded order relation on
   * List[T]. Stating that if xs < ys and ys < zs, then xs < zs.
   */
  axiom {
    forall xs : List[T], ys : List[T], zs : List[T] :: {decreasing(xs, ys
      decreasing(xs, ys) && decreasing(ys, zs) ==> decreasing(xs, zs)
  }
}
```

Figure 3.3: This code shows the definition of the well-founded order on the List[T] data type. It is the second part and complements Figure 3.2.

Pair Data Type

In section 7.4 of the book the function DivMod is introduced, for this function we need a type Pair. This type should contain two different types which can be individually set and returned. Dafny has built-in support for a Pair type while Viper does not have built-in support for tuples. DivMod should return two Unarys. The first argument of the pair should be the division of x by y and the second argument should be the remainder of the division.

```
function DivMod(x : Unary, y : Unary) : Pair[Unary, Unary]
    requires y != Zero()
    decreases UnaryToNat(x)
{
    Less(x, y) ? pair(Zero(), x)
        :
        let _ == (lemmaSubCorrect(x, y)) in
        let r == (DivMod(Sub(x, y), y)) in
        pair(Suc(getLeft(r)), getRight(r))
}
```

Figure 3.4: This function is the first function in the book where the Pair data type is needed.

After agreeing on the behavior of the Pair type, the question of how to implement the type in Viper was simple to answer. User-defined data types can be created using the domains and therefore the obvious choice is to model the pair type using a domain. With the help of a domain, we can specify the behavior of our new type clearly. By using axioms we can define which value will be returned by either *getLeft()* or *getRight()* as seen in Figure 3.5.

```
domain Pair[T, G] {
    function pair(T, G) : Pair[T, G]

    function getLeft(Pair[T, G]) : T
    function getRight(Pair[T, G]) : G

    axiom axgetLeft {
        forall x : T, y : G ::
        getLeft(pair(x, y)) == x
    }

    axiom axgetRight {
        forall x : T, y : G ::
        getRight(pair(x, y)) == y
    }
}
```

Figure 3.5: This domain in Viper models a Pair data type.

Asserting function

While proving lemmas it can be helpful to first write the proof body in methods. This is due to being able to write if and else clauses in methods, calling lemmas directly and assigning it to a variable rather than always using the let statements. This makes the whole body clearer to read and easier to understand. It also allows for easier debugging considering that we can introduce assumptions at multiple points in the program to focus on specific paths of the proof. With the help of such statements, we can completely disable a proof subtree. This means we are able to ignore a branch and try proving each proof tree branch individually.

The method in Figure 3.6 is an example of how the lemmas in Dafny of the book were first translated into methods in order to make it easier to prove the postconditions. However, at some point, the 1 to 1 translation of the methods to functions did not work anymore. We realized that at some points the functions do not leverage the result of other functions. Therefore, we had to assert certain equalities inside the function body explicitly in order for the Viper to actually check whether the assertion holds.

```
method lemmaInsertSameElementsM(y : Int, xs : List[Int], p : Int)
    ensures Project(Cons(y, xs), p) == Project(Insert(y, xs), p)
    decreases xs
{
    if (y==p){
        if (xs.isNil){

        }
        else {
            if (xs.value <= y){
                if (xs.value == p){
                    lemmaInsertSameElementsM(y, xs.tail, p)
                    assume false
                }
                else {
                    assert Project(Cons(y, xs), p) ==
                        Cons(y, Project(xs, p))
                    lemmaInsertSameElementsM(y, xs.tail, p)
                    assume false
                }
            }
            else {
                assume false
            }
        }
    }
    else {
        var a : Unit
        if (xs.isNil){
            assume false
        }
        else {
            var a : Unit
            a := lemmaDifferentElementsInserted(y, xs, p)
            assert Project(Insert(y, xs), p) == Project(xs, p)
            lemmaInsertSameElementsM(y, xs.tail, p)
            assert Project(Cons(y, xs), p) == Cons(y, Project(xs, p))

            assume false
        }
        a := lemmaDifferentElementsInserted(y, xs, p)
    }
}
```

Figure 3.6: This method shows the process of proving the lemma InsertSameElements in Viper. This is the equivalent proof to the one seen in Figure 3.8 where we prove it using a function.

In order to assert conditions inside the function bodies we used a new function called asserting. The definition of the function in Viper is shown in Figure 3.7. The most important part of this function is the precondition it requires. By requiring the input x to hold upon entry to the function it will trigger an error if this is not the case. With this function we can assert boolean expressions and enforce Viper to prove the assertions. This asserting of properties is important as it is often necessary to make Viper aware of these facts that are necessary to conclude proofs. We are not using the assert statement provided by Viper since we can not use statements in functions and are only allowed to use expressions. This reason encouraged us to create the asserting function.

```
function asserting(x : Bool) : Unit
    requires x
    decreases
{
    unit()
}
```

Figure 3.7: This function shows the asserting function used in this thesis.

Once it has checked the assertion it used it in the proof as well and was able to verify the postconditions in the functions. As seen in Figure 3.8 with the help of the asserting function and let expressions we are able to mimic the style of proofs that we would have written in methods.

```
function lemmaInsertSameElements(y : Int, xs : List[Int], p : Int) : Unit
    ensures Project(Cons(y, xs), p) == Project(Insert(y, xs), p)
    decreases xs
{
    (y == p) ? xs.isNil ? unit()
            :
            (xs.value <= y) ? (xs.value == p) ? let _0 ==
            (lemmaInsertSameElements(y, xs.tail, p)) in
                asserting(Project(Cons(y, xs), p) ==
                    Project(Insert(y, xs), p))
               :
                let _0 == (asserting(Project(Cons(y, xs), p) ==
                    Cons(y, Project(xs, p)))) in
                let _1 == (lemmaInsertSameElements(y, xs.tail, p)) in
                asserting(Project(Cons(y, xs.tail), p) ==
                    Project(Insert(y, xs.tail), p))
                :
                unit()
                :
                lemmaDifferentElementsInserted(y, xs, p)
}
```

Figure 3.8: This function contains the proof deduced in the method with the same name shown in Figure 3.6.

3.2 Bugs Found in Viper

Knowledge not used

```
function lemmaAtAppend(xs : List[Int], ys : List[Int], i : Int) : Unit
  requires i >= 0
  requires i < length(append(xs, ys))
  ensures let _ == (lemmaLengthAppend(xs.tail, ys)) in
          At(append(xs, ys), i) ==
          ((i < length(xs)) ? At(xs, i) : At(ys, i - length(xs)))
  decreases i
{
    (i==0) ? unit() : let _ == (lemmaLengthAppend(xs.tail, ys)) in
                        lemmaAtAppend(xs.tail, ys, i-1)
}
```

Figure 3.9: This is a lemma in Viper that needed further knowledge to prove the postcondition.

While translating the examples from Dafny to Viper the problem arose that certain knowledge needed to prove the postcondition of a function was not triggered and used by Viper. In this example 3.10 we need the additional knowledge of lemmaLengthAppend which should be triggered by the expression *(let _ == (lemmaLengthAppend(xs.tail, ys)) in At(append(xs, ys), i) == ((i ¡ length(xs)) ? At(xs, i) : At(ys, i - length(xs))))*.

There was a suspicion that this problem arose due to a bug in Viper Silicon. This suspicion was confirmed by the fact that Silicon could not prove the lemmas, but Carbon was. The reason for this behavior was Silicon did not instantiate the knowledge provided by the let expression. Silicon ignored the call to the lemma as long as the variable did not appear in the body. The knowledge was therefore only present once the variable _ assigned to the statement was mentioned in the body. After opening an issue [12] on the Viper GitHub repository the issue was resolved and knowledge from let statements will now be triggered even if the variable is not mentioned in the function body.

```
function TriggerLemma(x : Unit) : Unit
  decreases

function F67(x : Int, y : Int) : Int
  decreases

function L67() : Int
  decreases

function R67() : Int
  decreases

function lemmaLeftUnit67(x : Int) : Unit
  ensures F67(L67(), x) == L67()
  decreases

function lemmaRightUnit67(x : Int) : Unit
  ensures F67(x, R67()) == R67()
  decreases

function lemmaLEqualR67() : Unit
  ensures L67() == R67()
  decreases
{
  let _0 == (F67(L67(), R67())) in
  let _1 == (TriggerLemma(lemmaLeftUnit67(R67()))) in
  let _2 == (TriggerLemma(lemmaRightUnit67(L67()))) in
  unit()
}
```

Figure 3.10: This is a lemma in Viper that needed further knowledge to prove the postcondition.

Before the issue was resolved by the Viper development team, a workaround was implemented. A first fix implemented was to simply create a function called TriggerLemma. This function took an argument of type Unit and returned a Unit type. Due to this function call the lemma that was assigned in the let statement was used by the SMT solver. After the issue on GitHub was resolved all the TriggerLemma calls were removed from the final version of the code.

Decreases Keyword Positioning

```
import <decreases/int.vpr>

function sum(n : Int) : Int
    requires n >= 0
    ensures result == n*(n+1)/2
    decreases n
{
    ((n==0) ? 0 : n + sum(n-1))
}

method lemmasum(n : Int)
    //decreases n
    requires n >= 0
    ensures n != 0 ==> sum(n) == sum(n-1) + n
    decreases n
{}
```

Figure 3.11: This Figure shows a method and function written in Viper. The method lemmasum illustrates the problem with the decreases clause found in Viper.

Another problem encountered in Chapter 6 was that the ″decreases″ clause could only be used at the beginning of the method and not after the preconditions and postconditions. The error message generated is ″Verification aborted exceptionally″ and it is confusing. After carefully analysing the examples we could concluded that the message was generated due to the ″decreases″ clause being at the wrong position in the method specification.

This was a bug of Viper and after opening an Issue on the GitHub [13] page the problem was resolved and the decreases clause can now be written anywhere in the specification of the method.

3.3 Errors during translation

Abstract Loops

In Dafny we can use abstract loops. Abstract loops are loops where we do not need to provide a loop body, This can be seen in Figure 3.12. The verification language can assume the invariant is not violated and use the invariant and loop condition to prove further statements. In Viper we do not have a feature that allows the use of abstract loops. This made it very confusing at the beginning.

```
method Example0() {
  var x := 0;

  while x < 300
    invariant x % 2 == 0
}
```

Figure 3.12: This code example shows a code written in Dafny. It shows an abstract loop.

We created examples as seen in Figure 3.13 as we wanted to mimic the Dafny code as best as possible. We then realized it is not possible since we could prove assertions that should not be provable, as seen in Figure 3.14. Therefore, we started to write simple loop bodies for each of the loops that would not violate the invariant and would still allow us to continue with the translation of the examples.

```
method Loop11020(x : Int)
    requires x % 2 == 0
 {
    var y : Int
    y := x
    while(y < 300)
        invariant y % 2 == 0
    {

    }
 }
```

Figure 3.13: This code example shows a method written in Viper containing a while loop and an invariant. In this case the while loop does contain an empty loop body which is not sound in Viper.

```
method Loop11020(x : Int)
    requires x % 2 == 0
 {
    var y : Int
    y := x
    assume y < 300
    while(y < 300)
        invariant y % 2 == 0
    {
        y := y + 2
    }
    assert false
 }
```

Figure 3.14: This code shows the same loop again as in Figure 3.13. Here, a simple loop body solves the unsoundness of the method.

3.4 Chapter Discussion

The first chapter of part 1 introduces the basic ways of specifying and reasoning about inductively defined data structures. List are introduced and used to explain algebraic data structures in chapter 6. In this first chapter, there were a lot of initial challenges that had to be resolved. Those challenges included creating a translation for the lemma type discussed in section 3.1. After introducing the new data type List we also needed to find a way to create a well-founded order as discussed in section 3.1. We also encountered bugs in Viper during the translation of the 6th chapter. Those include the bugs discussed in section 3.2 and section 3.2.

Chapter 7 of the book follows up on inductively defined data types with the inductive representation of Unary numbers. Here we again use ADTs to define our data type Unary. In chapter 7 we encountered less problems as we have already resolved most of them in chapter 6. We still encountered one major challenge, this challenged included creating a Pair data type which we implemented as a domain type. This challenge is described in section 3.1.

```
adt Unary {
    Zero()
    Suc(pred: Unary)
}
```

Figure 3.15: This code shows the definition in Viper of the data type Unary.

Just as in Chapter 6 we also needed to define a well-founded order on the

Unary data type. Thanks to our already defined order for Lists the task of creating such an order was straight forward.

```
/**
 * Just like in chapter 6.1, we will define
 * the well-founded order of this data
 * type using a Domain.
 */
domain UnaryWellFoundedOrder {

    /**
     * This axiom tells us that for each element of
     * type Unary, there exists a
     * lower bound. The lowest bound for each
     * Unary element is zero().
     */
  axiom {
    forall y : Unary ::
    bounded(y)
  }

  /**
   * This axiom defines the order of the Unary elements.
   * As we see here, x.pred
   * is smaller than x. We could have also
   * defined to order to be
   * decreasing(x, Suc(x)).
   */
  axiom {
    forall x : Unary ::
    decreasing(x.pred, x)
  }

  /**
   * This axiom adds the transitivity of the Unary data type.
   */
  axiom {
    forall x : Unary, y : Unary, z : Unary ::
    {decreasing(x, y), decreasing(y, z)}
    decreasing(x, y) && decreasing(y, z) ==> decreasing(x, z)
  }
}
```

Figure 3.16: This code shows the definition of the well-founded order of the Unary data type using Viper domains.

Chapter 8 specifies and verifies two algorithms, both for sorting. The ADT

List which was created in Chapter 6 is reused in this chapter to educate the reader about sorting in verification-aware programming languages. The two sorting algorithms used in this chapter are Insertion Sort and Merge Sort.

While translating the pedagogical examples from Dafny to Viper a big challenge arose that took some time to solve. We realised that in Viper we needed to assert certain properties explicitly to make Viper aware of those facts that are necessary to conclude the proofs. The problem and the solution we came up with are further described in subsection 3.1.

Chapter 10 introduces us to the Dafny feature Invariants. These invariants relate to the characteristics of immutable data structures. The topic of invariants is covered in more detail later in chapter 11 as the state prior to loop iterations. This chapter did not provide too much difficulty in translating and there was no major feature or bug that needed further attention.

In imperative programming, one of the most prominent programming constructs is the loop. Chapter 11 focuses on the process of reasoning about loops using loop invariants, a feature that often poses challenges to beginners. This makes the code examples of chapter 11 very important.

In chapter 11 we encountered a problem regarding abstract loops. In Dafny, loop bodies do not have to be provided. We can prove methods with loops simply by analyzing the invariant and assuming it will hold at all times. Viper does not provide such a syntax and we need to specify a loop body at all times. This problem is further described in section 3.3.

Following the introduction of loops and loop invariants in Chapter 11, Chapter 12 brings into focus another important feature. This chapter transitions from recursively defined specifications to iteratively defined methods. An example of such a method can be seen in Figure 3.17.

```
method ComputeFib(n : Int) returns (x : Int)
    requires 0 <= n
    ensures 0 <= x
    ensures x == Fib(n)
    decreases
{
    x := 0
    var i : Int := 0
    var y : Int := 1
    var tmp : Int := x
    while(i != n)
        invariant 0 <= i && i <= n
        invariant x == Fib(i) && y == Fib(i + 1)
        decreases n - i
    {
        tmp := x
        x := y
        y := tmp + y
        i := i + 1
    }
}
```

Figure 3.17: This code shows a function in Viper that computes the Fibonacci sequence. It is one of the examples translated from chapter 12.

Chapter 4

Evaluation

In this chapter, we analyse and evaluate the code examples and exercises created in Viper. Those examples are compared to the ones written in Dafny. We look at the annotation overhead of the Viper files as well as the Dafny files. To evaluate the code, we will be analyzing the performance of Dafny and the Viper files and comparing them to each other. This will give us an intuition on which verification language runs more optimized.

4.1 Annotation Overhead

There are many reasons for different annotation overheads in Dafny and Viper. A few examples for such overheads are the decreases clause that has to be added to every function or method in Viper in order to check for termination. Another reason for such an overhead is also the automatic induction where we have to specify the induction step for Viper every time and Dafny is able to automatically generate simple induction proves.

Before looking at the results of the analysis it is helpful to give the reader a clear understanding of how the annotation was evaluated. To analyse the annotation overhead we created a Python script. The script analysed the code examples as follows. All the comments in the files were ignored, meaning all lines that started with ″ * ″, ″/**″ or ″ */″ were ignored. We also removed all the empty lines and did not count lines containing only ″″ or ″″. In this thesis we analysed the annotation overhead in two different settings. First we just counted all the code lines and compared the Dafny and Viper chapters individually to each others. In the second version we focused on the lemmas and their bodies. We counted the lines of code contained in the lemma bodies and compared the Dafny and Viper chapter to each other.

The motivation for this approach is that we wanted to see the effectiveness of the Dafny automatic induction. This type of induction is mostly applied in lemmas. By counting the lines of code in lemma bodies, we can assess the impact of the automatic induction in Dafny. We choose to analyse the total lines of code as well, as we can compare them to the lines of code in the lemma bodies. By this, we see whether the difference in lines needed to solve the exercises can be attributed to mostly the automatic induction or whether other factors need to be considered.

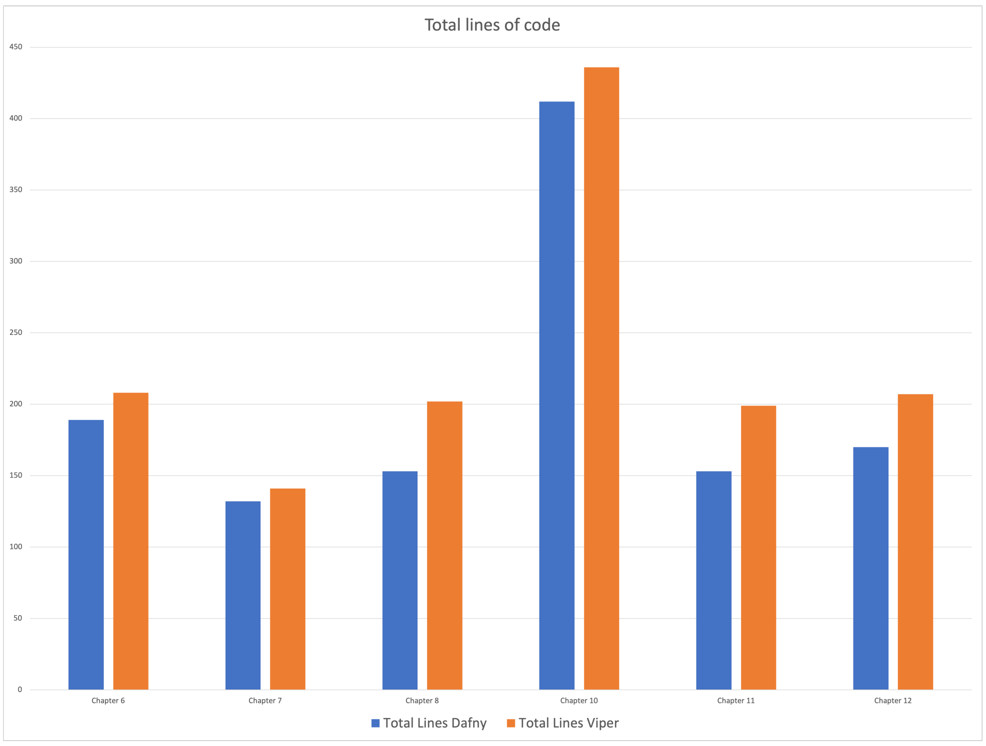

Figure 4.1: This graphic shows the total number of lines of code needed to complete each chapter. The blue bars show the total lines of code needed by Dafny and the orange lines show the total lines needed by Viper.

Figure 4.1 shows that Viper always needs more lines of code to encode the same amount of functions, methods, and lemmas. As we can see in Figure 4.2 Chapter 11 and Chapter 12 have no and almost no lemmas. Therefore, we can conclude that the difference in annotation has to come from something other than automatic induction. In chapter 11 Dafny uses a lot of simultaneous assignments. At the point of the code translation Viper did not support simultaneous assignments and we therefore had to assign the values on separate lines. Another reason for the higher amount of lines of code needed is the decreases clause for termination, as we have to add the termination measure to each Viper function or method.

Those different reasons account for the difference in lines of code needed for the different chapters.

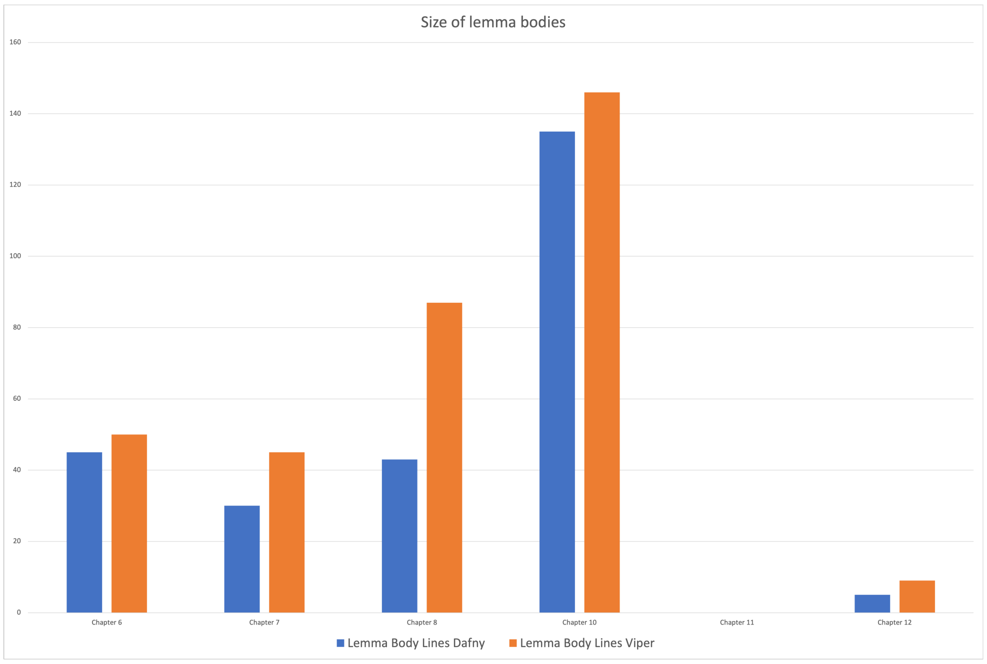

Figure 4.2: This graphic shows the lines of code in lemma bodies. The blue bars show the sizes of the lemma bodies of the Dafny lemmas, while the orange bars show the size of our created lemma bodies in Viper.

The lemma body is defined as the code of the lemma not including precondition, postcondition, and termination measure as well as the function signature. Evaluating Figure 4.2 we see a few interesting chapters. The difference in Chapter 6 and Chapter 7 between Dafny and Viper can be attributed to automatic induction being able to prove simple lemmas automatically in Dafny, while in Viper we have to provide the induction. In Chapter 8 there is a bigger difference. This is mostly due to four lemmas needing a lot of additional steps to be proven in Viper while Dafnys automatic induction is able to prove them with very little help. The difference in Chapter 10 is due to the same reasons as in Chapter 6 and Chapter 7. Chapter 11 does not contain any lemmas as it deals with loops and loop invariants. Chapter 12 only contains two lemmas, one of which can be solved by automatic induction in Dafny and the other that has to be proven manually in Dafny. Both of these lemmas need manual prove in Viper.

4.2 Performance Differences

The code was run on a PC using the Operating System Ubuntu 22.04.3 LTS, Graphics Card NVIDIA Corporation GA106 [GeForce RTX 3060], Processor AMD Ryzen 7 3700x-8core, RAM Corsair Vengeance LPX 2 x 16 GB 3200 MHz and Storage Samsung 970 EVO Plus 1 TB. We also used the commit 8fdb9d of Viper Silicon, commit 063de1 of Viper Carbon and the Dafny version 4.2.0 to analyze the performance of the different verification languages. In order for those verifiers to work a few other programs were needed, such as z3 satisfiability modulo theories (SMT) solver version 4.8.12 and the boogie modeling language 2.15.9.

In order to analyse the performance we created a Python script that used the function subprocess to invoke the different verifiers on the code. The script executed each file 10 times and in the end took the average of those executions.

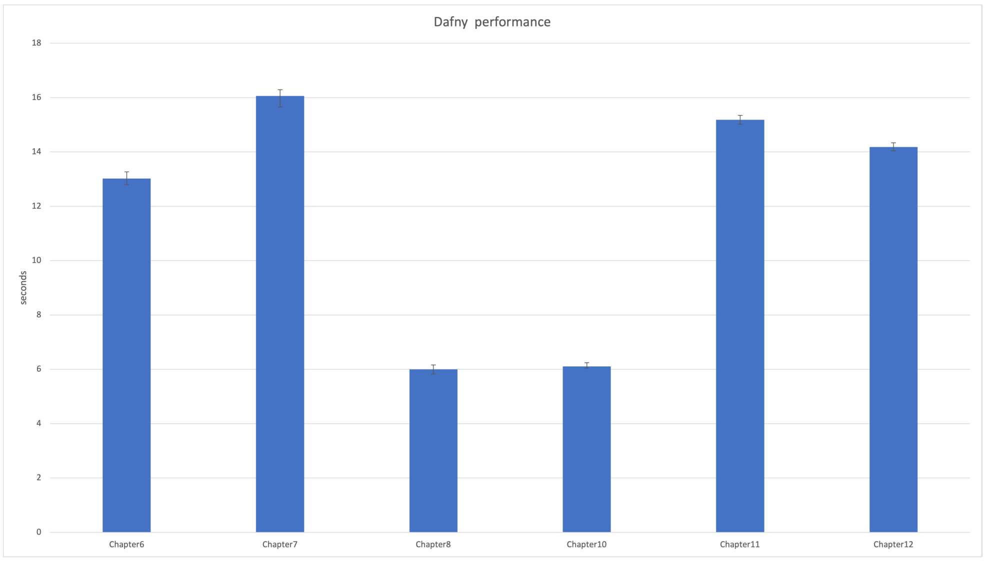

Figure 4.3: This graph shows the performance of the Dafny verifier. The y-Axis is measured in seconds. The error bar indicates the maximum and minimum value reached over the ten iterations.

This Figure 4.3 shows how much time each of the different chapters took to verify.

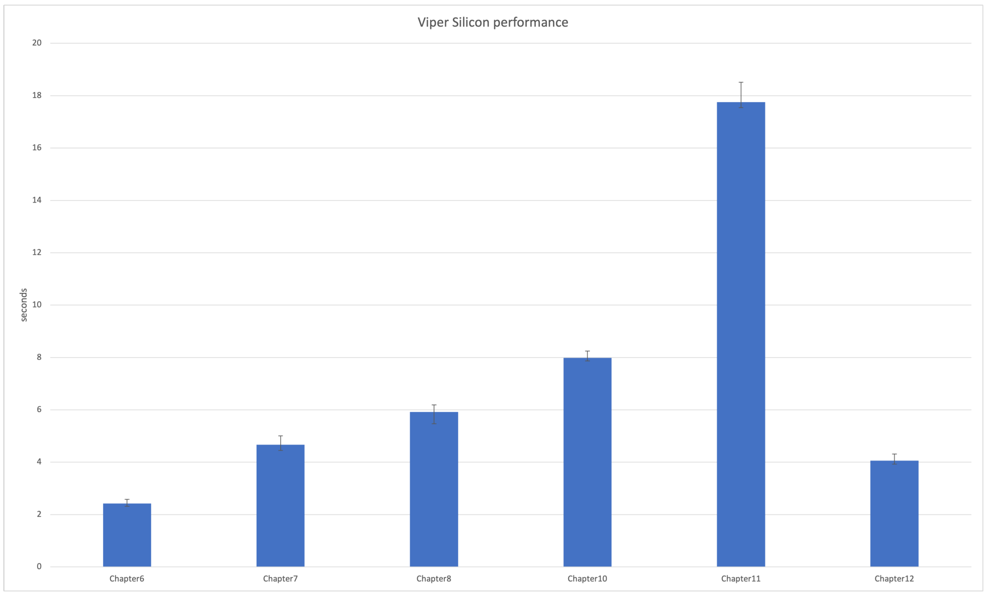

Figure 4.4: This graph shows the performance of the Viper Silicon verifier. The y-Axis is measured in seconds.The error bar indicates the maximum and minimum value reached over the ten iterations.

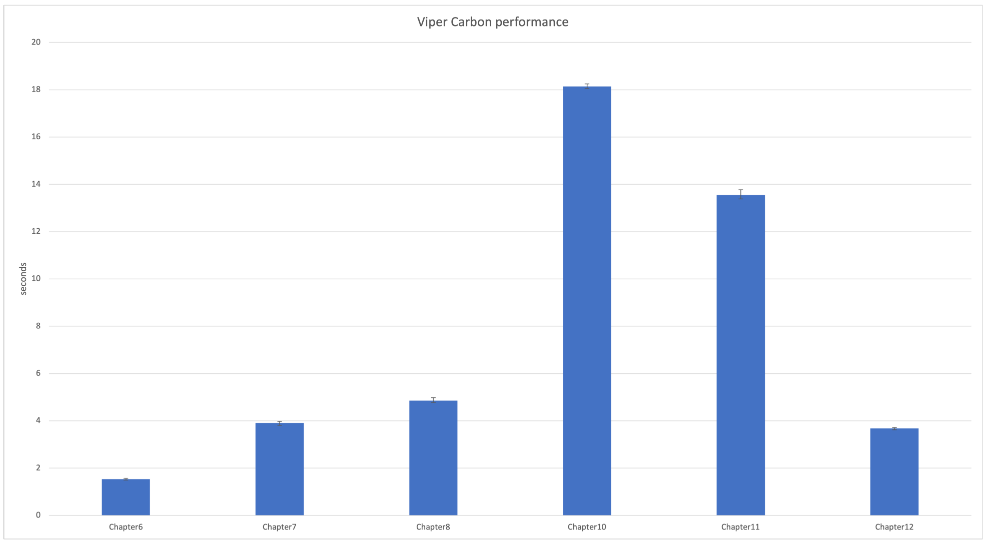

Figure 4.5: This graph shows the performance of the Viper Carbon verifier. The y-Axis is measured in seconds.The error bar indicates the maximum and minimum value reached over the ten iterations.

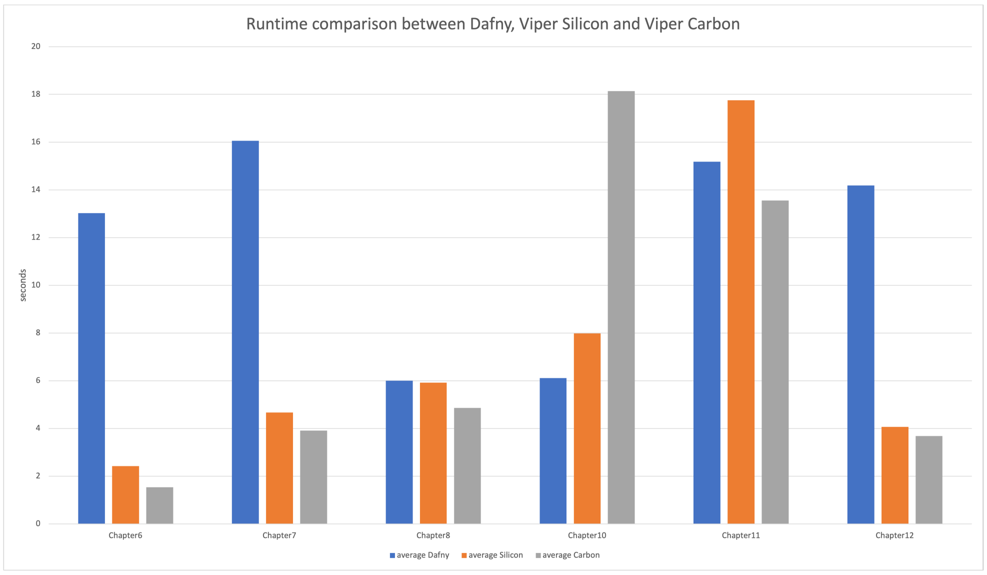

Figure 4.6: This graph shows the performance of all verifiers separately. The y-Axis is measured in seconds.

This Figure 4.6 shows the comparison of verification time between the different verifiers. A few interesting things to point out are related to chapter 6, chapter 7, chapter 10, and chapter 12. In the other chapters the three verifiers have similar verification times. Dafny seems to be slower in verifying chapter 6, chapter 7, and chapter 12, while Viper Carbon seems to be much slower in verifying chapter 10 than the other verifiers.

Unfortunately, we do not have a clear idea as to why the verifiers have so much slower verification times in certain chapters compared to the others. We can however speculate to why this might be the case.

As for most chapters, such as Chapter 6, Chapter 7 and Chapter 8 we provide a termination condition for all the functions written in Viper. In Dafny, all those conditions have to be generated. This might increase the performance overhead of Dafny to a degree in Chapter 6 and 7. It does however not explain why, in Chapter 8, Dafny seems to take an equal amount of time to verify the examples.

In chapter 10, Dafny and Silicon both take around the same time to verify the examples, while Carbon uses double the time. Dafny and Viper both have a similar amount of lines of code which supports the fact that Dafny and Silicon take similar time to verify. We understand the Silicon and Carbon backend not enough to reason about why Carbon takes double the time of Silicon.

In chapter 11 we have similar code line counts in both Dafny and Viper. Most of the methods and loops are also annotated with the same specifications. In this chapter, Dafny also annotates decreases clauses which removes the overhead generated by searching for an appropriate termination condition.

In Chapter 12, we again have termination conditions provided in the Viper code examples while they are omitted in the Dafny code. Other than the termination conditions the code is equivalent.

4.3 Possible Extensions

In this section, we suggest possible extensions that could be done to improve the Viper intermediate verification language. We would like to mention a few concepts of Dafny that would improve Viper.

Automatic Induction

As we have seen in the examples, Dafny supports automatic induction. This enables Dafny to prove postconditions automatically if the induction is basic. If such a tool is implemented in Viper it decreases the difficulty of proofing postconditions for the programmer. It reduces the complexity of certain proofs as it automates verification for simple recursive functions. It also decreases the annotation overhead for Viper code.

Default Termination Measures

Another useful implementation from our point of view would be to make termination proofs the default setting. Meaning Viper would always prove the total correctness of programs. It makes sense to prove total correctness always as we want to guarantee that a program will terminate. This is important as for most programs we need a guarantee for termination. Without the guarantee of termination most programs cannot be considered correct. This would also improve the annotation overhead of Viper as we would no longer have to specify the "decreases" clause every time we want Viper to prove total correctness. To aid with termination we could also include an automated termination condition generator that would try to automatically find termination conditions on which a termination proof could be done.

Lemmas

As Viper does not currently support a construct called lemmas, we would argue that implementing such a construct would help in verification. During this thesis lemmas were an essential part and helped a lot in proofing complex postconditions and splitting them up into separate parts. This would also help to differentiate between functions and lemmas, by not giving lemmas a return value and therefore never trying to use a lemma as a function. In general, lemmas increased the readability of the proofs and made it possible to state facts and equalities that should hold but needed proof.

If, Else conditions as expressions

In this thesis, we have used the ternary operator in almost all lemmas. It is more difficult to structure a function written with multiple ternary operators nested into each other and we can quickly lose the conditions on which we are branching. Therefore, it would make it more readable and clear if we could use if and else conditional branching in expressions as well. This would allow us to use if and else conditions not only in methods but also in functions.

Unnamed Binders

While proving postconditions, we used let bindings to introduce missing knowledge to the functions. In some cases, we had to use up to 10 let bindings, all of those bindings had to be named differently. This would ensure that we would be able to refer to them in the function body. In our case, however, we would have only needed to trigger the knowledge and not actually refer to it in the function body. Regarding this, we would not have needed distinct names for all let binders. A universal binder such as "_" would have sufficed for our purposes.

Calc Blocks

In chapter Background we have introduced the concept of calc blocks in Dafny 2.2. In the book Program Proofs they were used to write proof derivations steps. While translating those lemmas to Viper and proving the postcondition with the help of the derivation steps we used let statements. However, such calc blocks make the proof more readable and help the programmer keeping the proof clean. This can be seen by comparing the code in Dafny of Figure 4.7 to the code in Viper of Figure 4.8.

```
lemma ReverseCorrect<T>(xs: List<T>)
  ensures Reverse(xs) == SlowReverse(xs)
{
  calc {
    Reverse(xs);
  ==  // def. Reverse
    ReverseAux(xs, Nil);
  ==  { ReverseAuxSlowCorrect(xs, Nil); }
    Append(SlowReverse(xs), Nil);
  ==  { AppendNil(SlowReverse(xs)); }
    SlowReverse(xs);
  }
}
```

Figure 4.7: This code shows a lemma written in Dafny. To prove the postcondition a calc block was used.

```
function lemmaReverseCorrect(xs : List[Int]) : Unit
  ensures Reverse(xs) == SlowReverse(xs)
  decreases xs
{
  /**
   * In order for Viper to prove this lemma, we
   * again need to provide
   * the solver with some additional facts and lemmas.
   */
  xs.isNil ? unit() : let _0 ==
                        (lemmaReverseAuxSlowCorrect(xs, Nil())) in
                          let _1 ==
                        (lemmaAppendNil(SlowReverse(xs))) in
                          lemmaReverseCorrect(xs.tail)
}
```

Figure 4.8: This code shows the same lemma as in Figure 4.7 written in Viper.

Simultaneous Assignements

As we have seen in chapter Background subsection about simultaneous assignemnts 2.2, simultaneous assignments can help in reducing the lines needed to operate a switch between the values of two variables. In general, it makes sense to introduce simultaneous assignments. There are many cases where this would reduce the amount of code that has to be written. This can be seen in example 4.9 compared to the Viper version seen in Figure 4.10.

Fortunately, this has now already been implemented in Viper, and as of now, we are able to use simultaneous assignments [14].

```
method DivMod7() {
  var x, y;

  x, y := 0, 191;
  while 7 <= y
    invariant 0 <= y && 7 * x + y == 191
    decreases y
  {
    y := y - 7;
    x := x + 1;
  }
}
```

Figure 4.9: This method written in Dafny tries to illustrate the use of simultaneous assignemtns.

```
method Loops11200()
    decreases
{
    var x : Int
    var y : Int
    x := 0
    y := 191
    while(7 <= y)
        invariant 0 <= 7 && 7 * x + y == 191
        decreases y
    {
        y := y -7
        x := x + 1
    }
}
```

Figure 4.10: This method written in Viper is equivalent to the method of Figure 4.9. It shows the increased amount of lines needed in Viper due to Viper not supporting simultaneous assignments.

Chapter 5

Conclusion

This thesis handled 3 core goals. Those core goals are:

- Translating Pedagogical Verification code examples to Viper.
- Identifying examples that are difficult to implement in Viper.
- Comparing the annotation overhead and verification time between Viper and Dafny.

The main goal of this thesis was to develop code examples that help enhance the learning experience for people getting newly acquainted with program verification languages. For the first core goal we translated the examples and exercises from the book Program Proofs written in Dafny to Viper. The translation of the examples was mostly simple. It also included some implementations of features from Dafny in Viper that had no equivalent in Viper as described in section 3.1. Translating and solving the exercises took more time as certain proofs were extensive. At the end, all the material has been created. They are well documented to make them easy to understand and read, making them suitable pedagogical material.

While analysing and evaluating the annotation overhead between Dafny and Viper we created quantifiable numbers to show that Dafny has a slight advantage regarding the extra annotations that have to be made in order to verify code. We evaluated the performance of Dafny, Silicon and Carbon. By referring to Figure 4.6 we see that Viper Silicon mostly performs better or equally as good as Dafny or Carbon.

The core goals of this thesis regarding the translation of the examples and exercises, identifying examples that are difficult to implement in Viper as comparing the annotation overhead, verification time of Dafny and Viper, and creating pedagogical material for Viper have been met.

After the translation we can confidently say that at the moment, Dafny supports more features than Viper that make it more convenient to program in

Dafny. However, we have to mention that we translated examples specifically designed and tailored to the Dafny verification language and the result of our evaluation might therefore be biased towards the Dafny verification language.

Bibliography

[1] K. R. M. Leino, *Program Proofs*. MIT Press, 2023.

[2] Programming Methodology Group, ETH Zürich, "Viper Programming Language," https://www.pm.inf.ethz.ch/research/viper.html, online; accessed 05 July 2023.

[3] The dafny-lang community, "The Dafny Programming and Verification Language," https://dafny.org/, online; accessed 30 June 2023.

[4] Programming Methodology Group, ETH Zürich, "Gobra Programming Language," https://www.pm.inf.ethz.ch/research/gobra.html, online; accessed 31 July 2023.

[5] Programming Methodology Group, ETH Zürich, "Nagini programming language," https://www.pm.inf.ethz.ch/research/nagini.html, online; accessed 31 July 2023.

[6] Programming Methodology Group, ETH Zürich, "Viper github repository," https://github.com/viperproject/silver, online; accessed 23 April 2023.

[7] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[8] M. Häuser, "Use of symbolic execution for verification," *Technische Universität Kaiserslautern*.

[9] M. J. Frade and J. S. Pinto, "Verification conditions for source-level imperative programs," *Computer Science Review*, vol. 5, no. 3, pp. 252–277, 2011.

[10] K. Rustan M. Leino, "Code examples dafny program proofs," https://program-proofs.com/code.html, online; accessed 31 July 2023.

[11] Programming Methodology Group, ETH Zürich, "Viper tutorial," http://viper.ethz.ch/tutorial/, online; accessed 10 August 2023.

[12] Programming Methodology Group, ETH Zürich, "Viper github issue let statement," https://github.com/viperproject/silver/issues/688, online; accessed 23 April 2023.

[13] Programming Methodology Group, ETH Zürich, "Viper github decreases issue," https://github.com/viperproject/silver/issues/689, online; accessed 23 April 2023.

[14] Programming Methodology Group, ETH Zürich, "Viper pull request introducing simultaneous assignemnts," https://github.com/viperproject/silver/pull/685, online; accessed 10 September 2023.

Appendix

This section contains the code examples that are useful to have in the thesis but would disrupt the flow, if they would be place in the thesis.

```
/**
 * Viper does not have built-in support for
 * tuples (pairs, triples, ...), but we can
 * still define them via domains or ADTs.
 *
 * The type ‘Pair‘ takes two type parameters
 * and creates a data type pair out of
 * those two parameters.
 */
domain Pair[T, G] {
    /**
     * This is the constructor of a pair type.
     * The constructor takes two elements
     * of different types or the same type and
     * returns a pair element.
     */
    function pair(T, G) : Pair[T, G]

    /**
     * These functions define the destructors.
     * One is used to get the left element
     * or zero element of the pair and the
     * other one is used to get the right
     * element or one element of the pair.
     */
    function getLeft(Pair[T, G]) : T
    function getRight(Pair[T, G]) : G

    /**
     * This axiom defines the behavior of the
     * function get0. This defines that
     * get0 will always return the left element
     * of the tuple.
     */
    axiom axgetLeft {
        forall x : T, y : G ::
        getLeft(pair(x, y)) == x
    }

    /**
     * This second axiom defines the get1
     * function. This defines that get1
     * will always return the right element of
     * the tuple.
     */
    axiom axgetRight {
        forall x : T, y : G ::
        getRight(pair(x, y)) == y
    }
}
```

Figure 1: This Figure shows a new type Pair created in Viper using the domain concept.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Translating Pedagogical Verification Exercises to Viper.

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Frei	Benjamin

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Klingnau, 17.03.2023	B.Frei

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.